OPI
Lecture 22
Java generic types & classes
Java collection library

Kasper Østerbye

Carsten Schuermann

IT University Copenhagen

Motivation & Contents

- The primary purpose of generics is to provide a type safe library of collections.

- Usage examples from the collection library
- Autoboxing
- Co-variance and how to deal with it.
- Type constraints
- Collection hierarchy
- Implementation of generics
- Restrictions on type parameters

The simple homogeneous collection

Using generics, we can declare a List to be a list of Person objects:

```
List<Person> pl = new ArrayList<Person>();
```

This list is type safe:

```
pl.add( new Person(”Niels”) ); // OK
pl.add( new Fruitcake() ); // Not OK
Person p = pl.get(0); // OK, we know it contains
   Persons

pl.add( new Student() ); // OK, student is a
   subclass of Person
```

With the new autoboxing facility, we can also do:

```
List<Integer> il = new ArrayList<Integer>();

il.add(7); // OK
il.add( 4.5 ); // Not OK, double is not demoted

int x = il.get(0); // OK, it is a list of Integers
```

The new for-loop is quite useful in connection with generic collections:

```
for (int i : il ){ sum += i; } // OK, because we know
that il only contains Integers.
```

The List interface

```
interface List<E>{
    boolean add(E o);
    boolean remove(Object o);
    boolean contains(Object o);
    E get(int index);
    Iterator<E> iterator() ;
    List<E> subList(int fromIndex, int toIndex);
    …
}


List<Person> lp;
List<Vehicle> lv;
```

Formal type parameter E

Application/usage of type parameter E.
Attempting to add something which is not an E will cause an compile-time error.

Application/usage of type parameter E.
Because only elements of type E is added, we can ensure that something of type E is returned.

Application/usage of type parameter E.
The iterator of list is know to return only elements of type E

Application/usage of type parameter E.
Similarly, a sub-list is also a list of elements of type E

Actual type parameter Person.
List<Person> is said to be an *invocation* of List

HashMap

A map is a data structure which maps keys of some type to values of some possible other type.

```
Map<String,Person> nameMap =
   new HashMap<String,Person>();

nameMap.put(”Hans”, new Person(”Hans Ree”) );
nameMap.put(”Niels”, new Student(”Niels Puck”) );
Person p = nameMap.get(”Hans”);

Collection<Person> pc = nameMap.getValues();
```

```
// K is the type of the keys, V is the Value type
interface Map<K,V>{
   V put(K key, V value) ;
   V get(Object key);
   V remove(Object key);
   Collection<V> values() ;
   …
   interface Entry<K,V>{
      K getKey();
      V getValue();
      …
   }
   Set< Entry<K,V> > entrySet();
}
```

Hashmap – inner interfaces

The inner interface is considered a static member of the enclosing interface.

It is therefore not possible in the inner interface to refer to the type parameters of the outer interface.

The K,V in the inner interface are really new names introduced by the formal parameters in the Entry interface.

In the type instance Set< Entry<K, V> >, K, V refers to the parameters to Map, and are *actual* parameters for Entry, and applications of parameters from Map.

```
// K is the type of the keys, V is the Value type
interface Map<K,V>{
    V put(K key, V value) ;
    V get(Object key);
    V remove(Object key);
    Collection<V> values() ;
    …
    interface Entry<K,V>{
        K getKey();
        V getValue();
        …
    }
    Set< Entry<K,V> > entrySet();
}
```

Feeding tigers - 1

The problem of feeding tigers is a classic OO problem, it is related to using collections, but is more intuitive.

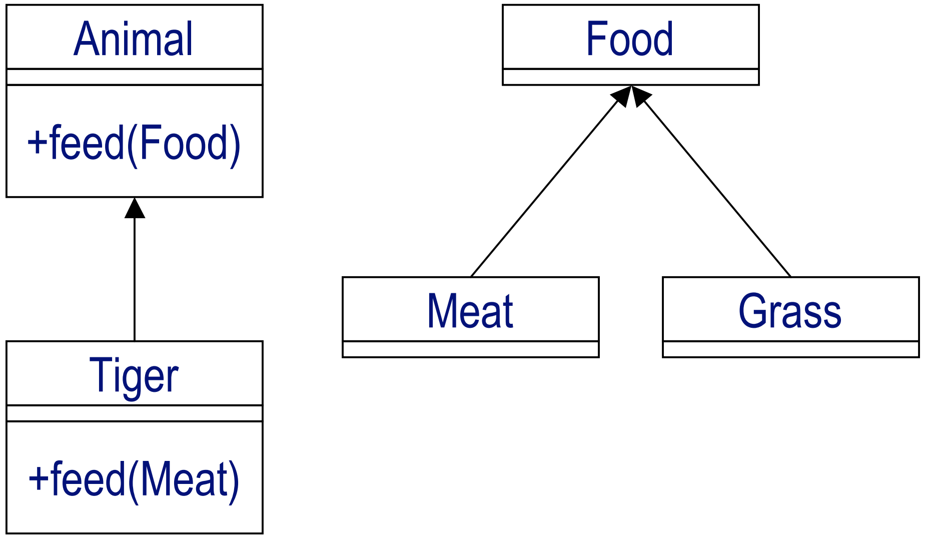
Assume we have the classes to the right. We then declare:

```
Animal a;
Tiger hobbes = new Tiger();
a = hobbes;
a.feed( new Grass() );
```

This is OK in Java.

Hobbes will be fed grass, and will not be happy.

1) feed in Tiger adds a new method and does not override the on in animal.
2) Even if it did, it would not be discovered until runtime that something was wrong.



The goal is to write a program which allows the compiler to make sure hobbes is only fed meat.

Feeding tigers - 2

```
class Animal<SomeFood>{
    private String name;
    Animal(String name){
        this.name = name;
    }
    void feed(SomeFood f){
        System.out.println(name + " was fed " + f.kind() );
    };
}

class Tiger extends Animal<Meat>{
    Tiger(String s){
        super(s);
    };
}
```

This nearly works:

- We cannot call the kind() method, as we do not know that SomeFood is of type Food.

This can be resolved by *bounding* the type parameter.

```
class Animal<SomeFood extends Food>{
    private String name;
    Animal(String name){
        this.name = name;
    }
    void feed(SomeFood f){
        System.out.println(name + " was fed " + f.kind() );
    };
}
```

Now, SomeFood is know to be a subtype of Food.

Feeding tigers - 3

Consider the code fragment:

```
Animal<Food> omnivore;
Tiger t;

t = new Tiger("Hobbes");
omnivore = t; // 1; Compile time error
omnivore.feed( new Grass() );
```

At 1) we get a compile time error, it is unsafe to assign a tiger to an omnivore.

The compiler now ensures we do not feed tigers grass.

Rule:

Sometype<X> and Sometype<Y> are always different types.

But then we cannot make a polymorphic variable?

```
Animal<?> a;
a = t; // OK.
a.feed( new Meat() ); // Not OK
a.feed( new Food() ); // Not OK
```

The idea is that the *wildcard* ”?” say that a is a variable which can refer to any animal which eats some kind of food – only we do not know which kind.

The tiger surely eats some kind of food, so the assignment is OK.

But we cannot feed a generic animal, as we do not know what food it eats.

Rule: Sometype<?> is supertype to all Sometype<X>

Feeding tigers - 4

Assume we want to write a method with two parameters, the animal to be fed, and the food to feed it with.

```
public static void feed(Animal<X> a, Food f){
    a.feed(f);
}
```

The X should be the same as the specific food provided.

```
public static void feed(Animal<X> a, X f){
    a.feed(f);
}
```

But both X'es are applications of a type variable, where should we declare it?

```
public static <X extends Food> void feed(Animal<X> a, X f){
    a.feed(f);
}
```

A call such as

```
feed( hobbes, new Meat() )
```

is legal, while other kinds of food are not accepted.

The compiler will deduce the right type for X. The compiler might produce mysterious error messages if it can not deduce it – that is, there is an error.

Summary

```
class MyClass <X extends Y>{
   X get(){…};
  void set(X x){…};
}
```

```
MyClass<?> something;
MyClass<Y> anything;
MyClass<Z> aZthing; // Z is a subclass of Y
```

	get / return	set/ as parameter	assignable from
something. (wildcard)	something.get() OK. return type is Y	something.set(x) not legal for any x	something = something; something = anything; something = aZthing
anything.	anything.get() OK. return type is Y	anything.set(x) OK. Legal for x instance of any subclass of Y	anything = anything
aZthing.	anZthing.get() OK. return type is Z	aZthing.set(z) OK. Legal for z instance of any subtype of Z	aZthing = aZthing

Back to collections

```
List<Person> lp = new ArrayList<Person>();
List<Student> ls;

ls.get(0).setGrade(9);

lp = ls; // Not allowed

lp.add( new Professor() );
```

It would be unsafe to alias a list of persons with a list of Students, as we can add non-Students to a person-list.

Hence the assignment is not allowed.

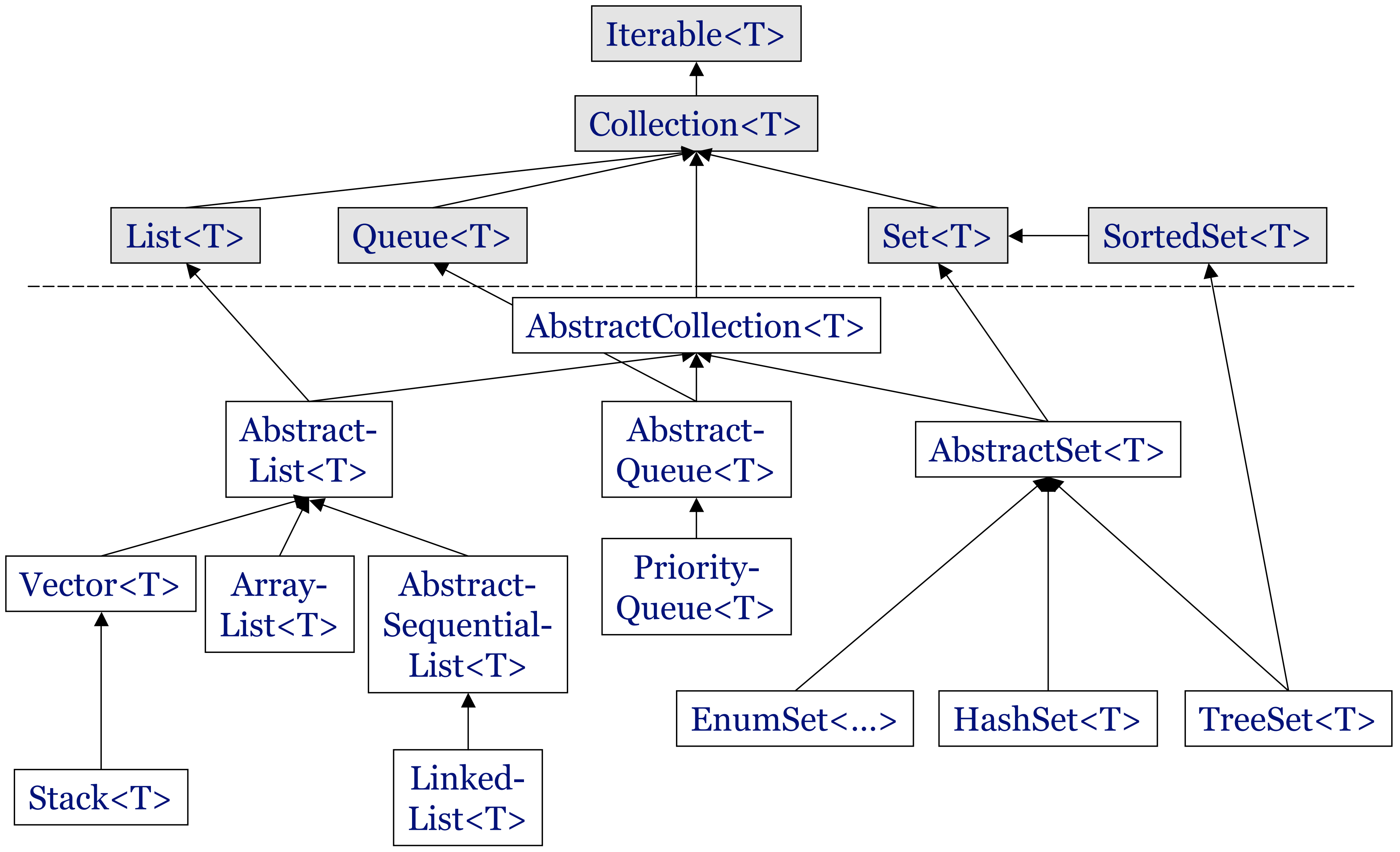
This corresponds to the omnivore and tiger.

```
List<? extends Person> lunknown;

lunknown = ls;

lunknown.add( new Person() );// Not allowed
```

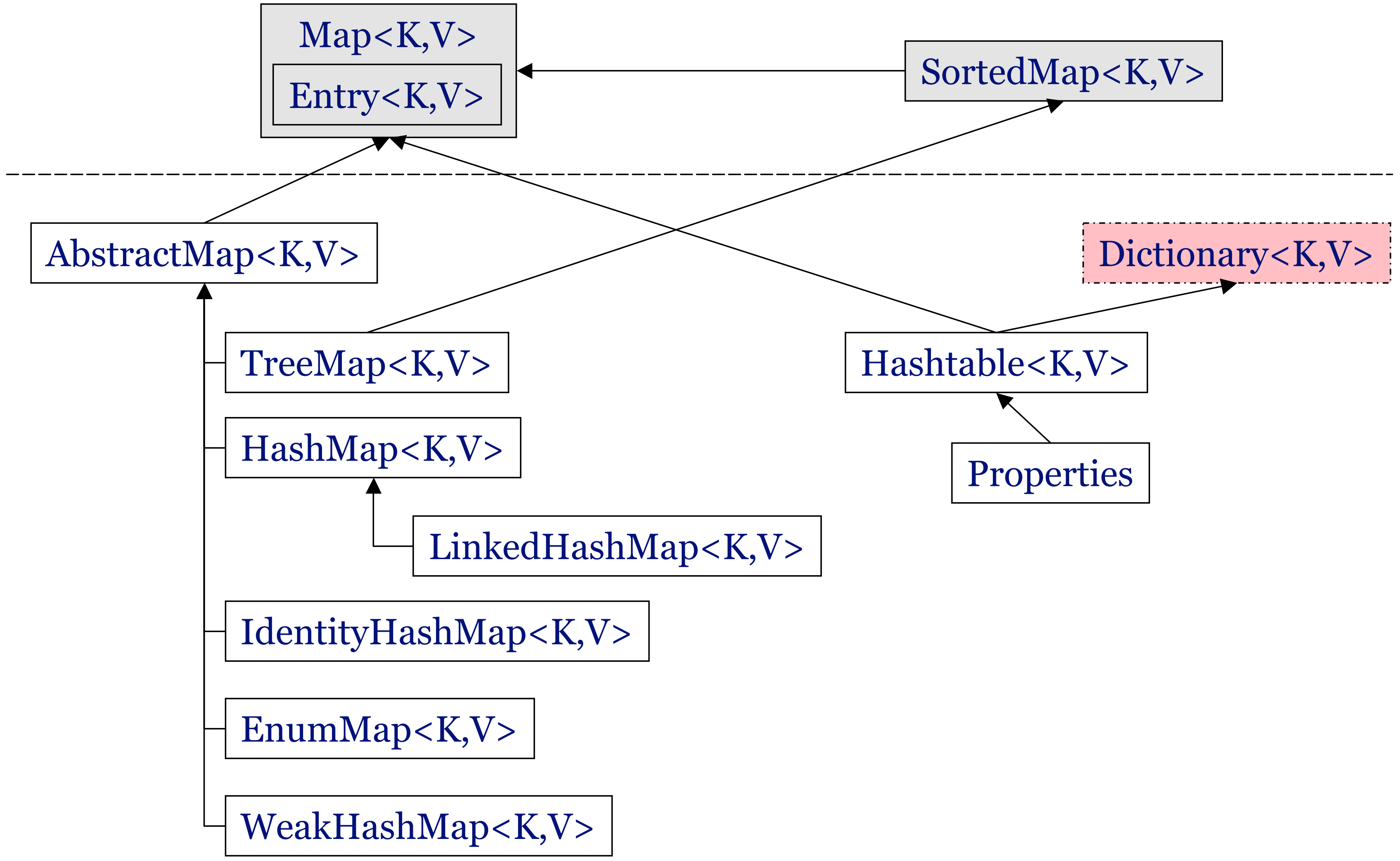
Similarly, a list of Persons *of some type* can be made. This list can be assigned a list of students.

But we cannot add any elements to it.

Java collection hierarchy - collections



Java collection hierarchy - maps



Iterator

- Iterable<T>
  - Iterator<T> iterator()
    classes implementing this interface can be used in the new for loop.
- Iterator<T>
  - boolean hasNext()
  - T next()
  - void remove() – optional !?

```
class Department implements Iterable<Employee>{
    Collection<Employee> emps;
    public Iterator<Employee> iterator(){
        return emps.iterator(); }
}

public static void main(String...args){
    Department d = new Department();
    d.emps = Arrays.asList(new Employee("Lars"),
        new Employee("Grethe"),new Employee("Hans") );

    for (Employee e: d)
        System.out.println( e.getName() );
}
```

? super T

The class Collections has a search method declared as:

```
public static <T> int binarySearch(
    List<? extends T> list,
    T key,
    Comparator<? super T> c)
```

It takes three arguments, a list to be searched, a key to look for, and a comparator.

```
interface Comparator<E>{
    int compare(T o1, T o2);
    …
}
```

<? super T> means T or any super type.

But why is that necessary?

Assume we have a List of Persons.

To do binary search, we need to compare the elements in the list.

Comparator<? extends Person> would mean we could give a comparator that works only for students. This is not OK.

Comparator<Person> is nearly OK, we must be able to compare any two persons.

But Comparator<LivingThing> should be OK too.

So we need something which is a person or more general.

Comparator<? super Person> is the way to say this.

Subclassing and generics

It is possible to use a generic class as super class.

```
class Tiger extends Animal<Meat> {…}
```

Tiger is really a subclass of Animal<Meat>, eg. aMeatEater = aTiger is OK.

It is possible to make a subclass which is itself generic

```
class Herbivore<T extends Plant> extends
    Animal<T>{…}
```

It is possible to make a generic subclass from a non-generic

```
class Animal<Food> extends LivingThing {…}
```

One can implement generic interfaces

```
class Cow extends Animal<Food> implements
    Producer<Milk> {…}
```

Implementation / Erasure

This is all implemented in the compiler, nothing is changed at run-time.

```
class MyClass <X extends Y>{
    X get(){…};
  void set(X x){…};
}
```

is translated into

```
class MyClass {
    Y get(){…};
    void set(Y x){…};
}
```

and

```
MyClass<Z> aZthing;
Z z = aZthing.get();
```

is translated into:

```
MyClass aZthing;
Z z = (Z) aZthing.get();
```

There is only one compiled version of MyClass.

Notice, the casts are inserted at the call place, not in the method itself.

Notice, the type parameters are erased at runtime. Therefore

```
List<Person> lp = new ArrayList<Person>();
List<Window> lw = new ArrayList<Window>();
```

The test `lp.getClass() == lw.getClass()` is true.

Raw types

The raw type of a generic is the type used without the type parameters.

```
List l; // Raw type
List<String> ls;
l = ls; // Allowed, no warning
l.add(new Integer(7) ); // 1; allowed, warning given
String s = ls.get(0); // 2; runtime error

ls = (List<String>)l; // 3; allowed, warning given
```

1) If we use add on a list with raw type, then a warning will be given by the compiler.
2) If we ignore the warning, we get a run time error (Integer cannot be cast to String)
3) We can cast l to a list of Strings. The compiler can not check this, and gives a warning. The type is erased at run time, and no runtime error is given.
   – But if the list contains a non string, a runtime error will occur when we read it.

Raw types are for compatibility with legacy code.

Do not use raw types unless it is the only option.

Restrictions on type parameters – 1 - new

A type parameter cannot be used in connection with new.

```
class Factory<X> {
    public X make(){
        return new X();
    }
}
```

The problem is that the type X does not exist at runtime. The above would be translated into:

```
class Factory {
    public Object make(){
        return new Object();
    }
}
```

This is not legal, nor what we wanted.

There is really no good solution to this problem.

The following works, but is cumbersome:

```
class Factory<T>{
    private Class<T> c;
    Factory(Class<T> c){
        this.c = c;
    }
    public T make(){
        try{
            return c.newInstance(); // reflection - inefficient
        }catch(Exception e)
            return null;
    }
}
Factory<Person> fp = new
                        Factory<Person>(Person.class);
Person p = fp.make();
```

Restrictions – 2 – arrays & static

One can use type parameters and parameterized types for array types, but not for array objects.

```
Animal<Meat>[] ama; // OK – type declaration
ama = new Animal<Meat>[10]; // Not allowed
```

The argument is complex (includes sending arrays as parameter).

Workaround:

Use ArrayList instead.

```
List< Animal<Meat> > ls
    = new ArrayList< Animal<Meat> >(10);
```

Array's are a primitive data structure which is used internally in the collection library ☹

Type parameters declared in the class header can not be used in static members.

```
class Animal<SomeFood extends Food>{
    private static SomeFood lastMeal; // NOT ALLOWED
    void feed(SomeFood f){
        lastMeal = f;
    };
    public static SomeFood lastEaten(){ // NOT ALLOWED
        return lastMeal;
    }
}

Animal<Meat> tiger;
Animal<Grass> cow;
tiger.feed( new Meat() );
Grass g = cow.lastEaten(); // would return meat
```

Restrictions - 3 –Overloadning & interfaces

One cannot overload on the type parameter

```
class Utilities {
   double measure(Animal<Grass> ag){…}
   double measure(Animal<Meat> am){…}
}
```

is NOT legal.

Because, there is only one compiled version of Animal.

Workaround 1 – add a dummy parameter.

```
class Utilities {
   double measure(Animal<Grass> ag, Grass g){…}
   double measure(Animal<Meat> am, Meat m){…}
}
```

Workaround 2 – make explicit classes

```
class Carnivore extends Animal<Meat>{}
class Herbivore extends Animal<Grass>{}
class Utilities {
    double measure(Herbivore h){…}
    double measure(Carnivore c){…}
}
```

One cannot implement the same interface for two different type parameters

```
class Cow extends Animal<Food>
    implements   Producer<Milk>,
                 Producer<Meat> {
    …
}
```

Workaround – same as 2 above

Collection concurrency issues

- A new library java.util.concurrent with BlockingQueue, ConcurrentMap iterfaces and several different implementations has been provided.
  - It also has several other heighly useful utilities for concurrency.